# Go语言反射（reflect），Golang反射（reflect）

< [Go语言加密通信](http://c.biancheng.net/view/vip_7357.html)[Go语言反射（reflection）](http://c.biancheng.net/view/4407.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/golang/reflect/_blank)

反射是指在程序运行期对程序本身进行访问和修改的能力。程序在编译时，变量被转换为内存地址，变量名不会被编译器写入到可执行部分。在运行程序时，程序无法获取自身的信息。  
  
支持反射的语言可以在程序编译期将变量的反射信息，如字段名称、类型信息、结构体信息等整合到可执行文件中，并给程序提供接口访问反射信息，这样就可以在程序运行期获取类型的反射信息，并且有能力修改它们。  
  
Go程序在运行期使用reflect包访问程序的反射信息。

#### 其它语言中的反射

C/C++ 语言没有支持反射功能，只能通过 typeid 提供非常弱化的程序运行时类型信息。Java、C# 等语言都支持完整的反射功能。  
  
Lua、JavaScript 类动态语言，由于其本身的语法特性就可以让代码在运行期访问程序自身的值和类型信息，因此不需要反射系统。  
  
Go 程序的反射系统无法获取到一个可执行文件空间中或者是一个包中的所有类型信息，需要配合使用标准库中对应的词法、语法解析器和抽象语法树（AST）对源码进行扫描后获得这些信息。

# Go语言反射（reflection）简述

< [Go语言反射](http://c.biancheng.net/golang/reflect/)[Go语言反射规则浅析](http://c.biancheng.net/view/5131.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

反射（reflection）是在 [Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) 出现后迅速流行起来的一种概念，通过反射可以获取丰富的类型信息，并可以利用这些类型信息做非常灵活的工作。  
  
大多数现代的高级语言都以各种形式支持反射功能，反射是把双刃剑，功能强大但代码可读性并不理想，若非必要并不推荐使用反射。  
  
下面我们就来将介绍一下反射在Go语言中的具体体现以及反射的基本使用方法。

## 反射的基本概念

Go语言提供了一种机制在运行时更新和检查变量的值、调用变量的方法和变量支持的内在操作，但是在编译时并不知道这些变量的具体类型，这种机制被称为反射。反射也可以让我们将类型本身作为第一类的值类型处理。  
  
反射是指在程序运行期对程序本身进行访问和修改的能力，程序在编译时变量被转换为内存地址，变量名不会被编译器写入到可执行部分，在运行程序时程序无法获取自身的信息。  
  
支持反射的语言可以在程序编译期将变量的反射信息，如字段名称、类型信息、结构体信息等整合到可执行文件中，并给程序提供接口访问反射信息，这样就可以在程序运行期获取类型的反射信息，并且有能力修改它们。  
  
C/[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank)语言没有支持反射功能，只能通过 typeid 提供非常弱化的程序运行时类型信息；Java、[C#](http://c.biancheng.net/csharp/" \t "http://c.biancheng.net/view/_blank) 等语言都支持完整的反射功能；Lua、[JavaScript](http://c.biancheng.net/js/" \t "http://c.biancheng.net/view/_blank) 类动态语言，由于其本身的语法特性就可以让代码在运行期访问程序自身的值和类型信息，因此不需要反射系统。  
  
Go语言程序的反射系统无法获取到一个可执行文件空间中或者是一个包中的所有类型信息，需要配合使用标准库中对应的词法、语法解析器和抽象语法树（AST）对源码进行扫描后获得这些信息。  
  
Go语言提供了 reflect 包来访问程序的反射信息。

## reflect 包

Go语言中的反射是由 reflect 包提供支持的，它定义了两个重要的类型 Type 和 Value 任意接口值在反射中都可以理解为由 reflect.Type 和 reflect.Value 两部分组成，并且 reflect 包提供了 reflect.TypeOf 和 reflect.ValueOf 两个函数来获取任意对象的 Value 和 Type。

## 反射的类型对象（reflect.Type）

在Go语言程序中，使用 reflect.TypeOf() 函数可以获得任意值的类型对象（reflect.Type），程序通过类型对象可以访问任意值的类型信息，下面通过示例来理解获取类型对象的过程：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var a int
8. typeOfA := reflect.TypeOf(a)
9. fmt.Println(typeOfA.Name(), typeOfA.Kind())
10. }

运行结果如下：

int  int

代码说明如下：

* 第 9 行，定义一个 int 类型的变量。
* 第 10 行，通过 reflect.TypeOf() 取得变量 a 的类型对象 typeOfA，类型为 reflect.Type()。
* 第 11 行中，通过 typeOfA 类型对象的成员函数，可以分别获取到 typeOfA 变量的类型名为 int，种类（Kind）为 int。

## 反射的类型（Type）与种类（Kind）

在使用反射时，需要首先理解类型（Type）和种类（Kind）的区别。编程中，使用最多的是类型，但在反射中，当需要区分一个大品种的类型时，就会用到种类（Kind）。例如需要统一判断类型中的指针时，使用种类（Kind）信息就较为方便。

#### 1) 反射种类（Kind）的定义

Go语言程序中的类型（Type）指的是系统原生数据类型，如 int、string、bool、float32 等类型，以及使用 type 关键字定义的类型，这些类型的名称就是其类型本身的名称。例如使用 type A struct{} 定义结构体时，A 就是 struct{} 的类型。  
  
种类（Kind）指的是对象归属的品种，在 reflect 包中有如下定义：

1. type Kind uint
2. const (
3. Invalid Kind = iota // 非法类型
4. Bool // 布尔型
5. Int // 有符号整型
6. Int8 // 有符号8位整型
7. Int16 // 有符号16位整型
8. Int32 // 有符号32位整型
9. Int64 // 有符号64位整型
10. Uint // 无符号整型
11. Uint8 // 无符号8位整型
12. Uint16 // 无符号16位整型
13. Uint32 // 无符号32位整型
14. Uint64 // 无符号64位整型
15. Uintptr // 指针
16. Float32 // 单精度浮点数
17. Float64 // 双精度浮点数
18. Complex64 // 64位复数类型
19. Complex128 // 128位复数类型
20. Array // 数组
21. Chan // 通道
22. Func // 函数
23. Interface // 接口
24. Map // 映射
25. Ptr // 指针
26. Slice // 切片
27. String // 字符串
28. Struct // 结构体
29. UnsafePointer // 底层指针
30. )

Map、Slice、Chan 属于引用类型，使用起来类似于指针，但是在种类常量定义中仍然属于独立的种类，不属于 Ptr。type A struct{} 定义的结构体属于 Struct 种类，\*A 属于 Ptr。

#### 2) 从类型对象中获取类型名称和种类

Go语言中的类型名称对应的反射获取方法是 reflect.Type 中的 Name() 方法，返回表示类型名称的字符串；类型归属的种类（Kind）使用的是 reflect.Type 中的 Kind() 方法，返回 reflect.Kind 类型的常量。  
  
下面的代码中会对常量和结构体进行类型信息获取。

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. // 定义一个Enum类型
7. type Enum int
8. const (
9. Zero Enum = 0
10. )
11. func main() {
12. // 声明一个空结构体
13. type cat struct {
14. }
15. // 获取结构体实例的反射类型对象
16. typeOfCat := reflect.TypeOf(cat{})
17. // 显示反射类型对象的名称和种类
18. fmt.Println(typeOfCat.Name(), typeOfCat.Kind())
19. // 获取Zero常量的反射类型对象
20. typeOfA := reflect.TypeOf(Zero)
21. // 显示反射类型对象的名称和种类
22. fmt.Println(typeOfA.Name(), typeOfA.Kind())
23. }

运行结果如下：

cat struct  
Enum int

代码说明如下：

* 第 17 行，声明结构体类型 cat。
* 第 20 行，将 cat 实例化，并且使用 reflect.TypeOf() 获取被实例化后的 cat 的反射类型对象。
* 第 22 行，输出 cat 的类型名称和种类，类型名称就是 cat，而 cat 属于一种结构体种类，因此种类为 struct。
* 第 24 行，Zero 是一个 Enum 类型的常量。这个 Enum 类型在第 9 行声明，第 12 行声明了常量。如没有常量也不能创建实例，通过 reflect.TypeOf() 直接获取反射类型对象。
* 第 26 行，输出 Zero 对应的类型对象的类型名和种类。

## 指针与指针指向的元素

Go语言程序中对指针获取反射对象时，可以通过 reflect.Elem() 方法获取这个指针指向的元素类型，这个获取过程被称为取元素，等效于对指针类型变量做了一个\*操作，代码如下：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. // 声明一个空结构体
8. type cat struct {
9. }
10. // 创建cat的实例
11. ins := &cat{}
12. // 获取结构体实例的反射类型对象
13. typeOfCat := reflect.TypeOf(ins)
14. // 显示反射类型对象的名称和种类
15. fmt.Printf("name:'%v' kind:'%v'\n", typeOfCat.Name(), typeOfCat.Kind())
16. // 取类型的元素
17. typeOfCat = typeOfCat.Elem()
18. // 显示反射类型对象的名称和种类
19. fmt.Printf("element name: '%v', element kind: '%v'\n", typeOfCat.Name(), typeOfCat.Kind())
20. }

运行结果如下：

name:'' kind:'ptr'  
element name: 'cat', element kind: 'struct'

代码说明如下：

* 第 13 行，创建了 cat 结构体的实例，ins 是一个 \*cat 类型的指针变量。
* 第 15 行，对指针变量获取反射类型信息。
* 第 17 行，输出指针变量的类型名称和种类。Go语言的反射中对所有指针变量的种类都是 Ptr，但需要注意的是，指针变量的类型名称是空，不是 \*cat。
* 第 19 行，取指针类型的元素类型，也就是 cat 类型。这个操作不可逆，不可以通过一个非指针类型获取它的指针类型。
* 第 21 行，输出指针变量指向元素的类型名称和种类，得到了 cat 的类型名称（cat）和种类（struct）。

## 使用反射获取结构体的成员类型

任意值通过 reflect.TypeOf() 获得反射对象信息后，如果它的类型是结构体，可以通过反射值对象 reflect.Type 的 NumField() 和 Field() 方法获得结构体成员的详细信息。  
  
与成员获取相关的 reflect.Type 的方法如下表所示。

|  |  |
| --- | --- |
| 结构体成员访问的方法列表 | |
| **方法** | **说明** |
| Field(i int) StructField | 根据索引返回索引对应的结构体字段的信息，当值不是结构体或索引超界时发生宕机 |
| NumField() int | 返回结构体成员字段数量，当类型不是结构体或索引超界时发生宕机 |
| FieldByName(name string) (StructField, bool) | 根据给定字符串返回字符串对应的结构体字段的信息，没有找到时 bool 返回 false，当类型不是结构体或索引超界时发生宕机 |
| FieldByIndex(index []int) StructField | 多层成员访问时，根据 []int 提供的每个结构体的字段索引，返回字段的信息，没有找到时返回零值。当类型不是结构体或索引超界时发生宕机 |
| FieldByNameFunc(match func(string) bool) (StructField,bool) | 根据匹配函数匹配需要的字段，当值不是结构体或索引超界时发生宕机 |

#### 1) 结构体字段类型

reflect.Type 的 Field() 方法返回 StructField 结构，这个结构描述结构体的成员信息，通过这个信息可以获取成员与结构体的关系，如偏移、索引、是否为匿名字段、结构体标签（StructTag）等，而且还可以通过 StructField 的 Type 字段进一步获取结构体成员的类型信息。  
  
StructField 的结构如下：

1. type StructField struct {
2. Name string // 字段名
3. PkgPath string // 字段路径
4. Type Type // 字段反射类型对象
5. Tag StructTag // 字段的结构体标签
6. Offset uintptr // 字段在结构体中的相对偏移
7. Index []int // Type.FieldByIndex中的返回的索引值
8. Anonymous bool // 是否为匿名字段
9. }

字段说明如下：

* Name：为字段名称。
* PkgPath：字段在结构体中的路径。
* Type：字段本身的反射类型对象，类型为 reflect.Type，可以进一步获取字段的类型信息。
* Tag：结构体标签，为结构体字段标签的额外信息，可以单独提取。
* Index：FieldByIndex 中的索引顺序。
* Anonymous：表示该字段是否为匿名字段。

#### 2) 获取成员反射信息

下面代码中，实例化一个结构体并遍历其结构体成员，再通过 reflect.Type 的 FieldByName() 方法查找结构体中指定名称的字段，直接获取其类型信息。  
  
反射访问结构体成员类型及信息：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. // 声明一个空结构体
8. type cat struct {
9. Name string
10. // 带有结构体tag的字段
11. Type int `json:"type" id:"100"`
12. }
13. // 创建cat的实例
14. ins := cat{Name: "mimi", Type: 1}
15. // 获取结构体实例的反射类型对象
16. typeOfCat := reflect.TypeOf(ins)
17. // 遍历结构体所有成员
18. for i := 0; i < typeOfCat.NumField(); i++ {
19. // 获取每个成员的结构体字段类型
20. fieldType := typeOfCat.Field(i)
21. // 输出成员名和tag
22. fmt.Printf("name: %v tag: '%v'\n", fieldType.Name, fieldType.Tag)
23. }
24. // 通过字段名, 找到字段类型信息
25. if catType, ok := typeOfCat.FieldByName("Type"); ok {
26. // 从tag中取出需要的tag
27. fmt.Println(catType.Tag.Get("json"), catType.Tag.Get("id"))
28. }
29. }

代码输出如下：

name: Name  tag: ''  
name: Type  tag: 'json:"type" id:"100"'  
type 100

代码说明如下：

* 第 10 行，声明了带有两个成员的 cat 结构体。
* 第 13 行，Type 是 cat 的一个成员，这个成员类型后面带有一个以 ` 开始和结尾的字符串。这个字符串在Go语言中被称为 Tag（标签）。一般用于给字段添加自定义信息，方便其他模块根据信息进行不同功能的处理。
* 第 16 行，创建 cat 实例，并对两个字段赋值。结构体标签属于类型信息，无须且不能赋值。
* 第 18 行，获取实例的反射类型对象。
* 第 20 行，使用 reflect.Type 类型的 NumField() 方法获得一个结构体类型共有多少个字段。如果类型不是结构体，将会触发宕机错误。
* 第 22 行，reflect.Type 中的 Field() 方法和 NumField 一般都是配对使用，用来实现结构体成员的遍历操作。
* 第 24 行，使用 reflect.Type 的 Field() 方法返回的结构不再是 reflect.Type 而是 StructField 结构体。
* 第 27 行，使用 reflect.Type 的 FieldByName() 根据字段名查找结构体字段信息，catType 表示返回的结构体字段信息，类型为 StructField，ok 表示是否找到结构体字段的信息。
* 第 29 行中，使用 StructField 中 Tag 的 Get() 方法，根据 Tag 中的名字进行信息获取。

## 结构体标签（Struct Tag）

通过 reflect.Type 获取结构体成员信息 reflect.StructField 结构中的 Tag 被称为结构体标签（StructTag）。结构体标签是对结构体字段的额外信息标签。  
  
JSON、BSON 等格式进行序列化及对象关系映射（Object Relational Mapping，简称 ORM）系统都会用到结构体标签，这些系统使用标签设定字段在处理时应该具备的特殊属性和可能发生的行为。这些信息都是静态的，无须实例化结构体，可以通过反射获取到。

#### 1) 结构体标签的格式

Tag 在结构体字段后方书写的格式如下：

`key1:"value1" key2:"value2"`

结构体标签由一个或多个键值对组成；键与值使用冒号分隔，值用双引号括起来；键值对之间使用一个空格分隔。

#### 2) 从结构体标签中获取值

StructTag 拥有一些方法，可以进行 Tag 信息的解析和提取，如下所示：

* func (tag StructTag) Get(key string) string：根据 Tag 中的键获取对应的值，例如`key1:"value1" key2:"value2"`的 Tag 中，可以传入“key1”获得“value1”。
* func (tag StructTag) Lookup(key string) (value string, ok bool)：根据 Tag 中的键，查询值是否存在。

#### 3) 结构体标签格式错误导致的问题

编写 Tag 时，必须严格遵守键值对的规则。结构体标签的解析代码的容错能力很差，一旦格式写错，编译和运行时都不会提示任何错误，示例代码如下：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. type cat struct {
8. Name string
9. Type int `json: "type" id:"100"`
10. }
11. typeOfCat := reflect.TypeOf(cat{})
12. if catType, ok := typeOfCat.FieldByName("Type"); ok {
13. fmt.Println(catType.Tag.Get("json"))
14. }
15. }

运行上面的代码会输出一个空字符串，并不会输出期望的 type。  
  
代码第 11 行中，在 json: 和 "type" 之间增加了一个空格，这种写法没有遵守结构体标签的规则，因此无法通过 Tag.Get 获取到正确的 json 对应的值。这个错误在开发中非常容易被疏忽，造成难以察觉的错误。所以将第 12 行代码修改为下面的样子，则可以正常打印。

1. type cat struct {
2. Name string
3. Type int `json:"type" id:"100"`
4. }

运行结果如下：

type

# Go语言反射规则浅析

< [Go语言反射（reflection）](http://c.biancheng.net/view/4407.html)[反射——性能和灵活性的双刃剑](http://c.biancheng.net/view/vip_7358.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

反射是众多编程语言中的一个非常实用的功能，它是一种能够自描述、自控制的应用，Go语言也对反射提供了友好的支持。  
  
Go语言中使用反射可以在编译时不知道类型的情况下更新变量，在运行时查看值、调用方法以及直接对他们的布局进行操作。  
  
由于反射是建立在类型系统（type system）上的，所以我们先来复习一下Go语言中的类型。

## Go语言中的类型

Go语言是一门静态类型的语言，每个变量都有一个静态类型，类型在编译的时候确定下来。

type MyInt int  
  
var i int  
var j MyInt

变量 i 的类型是 int，变量 j 的类型是 MyInt，虽然它们有着相同的基本类型，但静态类型却不一样，在没有类型转换的情况下，它们之间无法互相赋值。  
  
接口是一个重要的类型，它意味着一个确定的方法集合，一个接口变量可以存储任何实现了接口的方法的具体值（除了接口本身），例如 io.Reader 和 io.Writer：

// Reader is the interface that wraps the basic Read method.  
type Reader interface {  
    Read(p []byte) (n int, err error)  
}  
  
// Writer is the interface that wraps the basic Write method.  
type Writer interface {  
    Write(p []byte) (n int, err error)  
}

如果一个类型声明实现了 Reader（或 Writer）方法，那么它便实现了 io.Reader（或 io.Writer），这意味着一个 io.Reader 的变量可以持有任何一个实现了 Read 方法的的类型的值。

var r io.Reader  
r = os.Stdin  
r = bufio.NewReader(r)  
r = new(bytes.Buffer)  
// and so on

必须要弄清楚的一点是，不管变量 r 中的具体值是什么，r 的类型永远是 io.Reader，由于Go语言是静态类型的，r 的静态类型就是 io.Reader。  
  
在接口类型中有一个极为重要的例子——空接口：

interface{}

它表示了一个空的方法集，一切值都可以满足它，因为它们都有零值或方法。  
  
有人说Go语言的接口是动态类型，这是错误的，它们都是静态类型，虽然在运行时中，接口变量存储的值也许会变，但接口变量的类型是不会变的。我们必须精确地了解这些，因为反射与接口是密切相关的。  
  
关于接口我们就介绍到这里，下面我们看看Go语言的反射三定律。

## 反射第一定律：反射可以将“接口类型变量”转换为“反射类型对象”

注：这里反射类型指 reflect.Type 和 reflect.Value。

从使用方法上来讲，反射提供了一种机制，允许程序在运行时检查接口变量内部存储的 (value, type) 对。  
  
在最开始，我们先了解下 reflect 包的两种类型 Type 和 Value，这两种类型使访问接口内的数据成为可能，它们对应两个简单的方法，分别是 reflect.TypeOf 和 reflect.ValueOf，分别用来读取接口变量的 reflect.Type 和 reflect.Value 部分。  
  
当然，从 reflect.Value 也很容易获取到 reflect.Type，目前我们先将它们分开。  
  
首先，我们下看 reflect.TypeOf：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var x float64 = 3.4
8. fmt.Println("type:", reflect.TypeOf(x))
9. }

运行结果如下：

type: float64

大家可能会疑惑，为什么没看到接口？这段代码看起来只是把一个 float64 类型的变量 x 传递给 reflect.TypeOf 并没有传递接口。其实在 reflect.TypeOf 的函数签名里包含一个空接口：

// TypeOf returns the reflection Type of the value in the interface{}.  
func TypeOf(i interface{}) Type

我们调用 reflect.TypeOf(x) 时，x 被存储在一个空接口变量中被传递过去，然后 reflect.TypeOf 对空接口变量进行拆解，恢复其类型信息。  
  
函数 reflect.ValueOf 也会对底层的值进行恢复：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var x float64 = 3.4
8. fmt.Println("value:", reflect.ValueOf(x))
9. }

运行结果如下：

value: 3.4

类型 reflect.Type 和 reflect.Value 都有很多方法，我们可以检查和使用它们，这里我们举几个例子。  
  
类型 reflect.Value 有一个方法 Type()，它会返回一个 reflect.Type 类型的对象。  
  
Type 和 Value 都有一个名为 Kind 的方法，它会返回一个常量，表示底层数据的类型，常见值有：Uint、Float64、Slice 等。  
  
Value 类型也有一些类似于 Int、Float 的方法，用来提取底层的数据：

* Int 方法用来提取 int64
* Float 方法用来提取 float64，示例代码如下：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var x float64 = 3.4
8. v := reflect.ValueOf(x)
9. fmt.Println("type:", v.Type())
10. fmt.Println("kind is float64:", v.Kind() == reflect.Float64)
11. fmt.Println("value:", v.Float())
12. }

运行结果如下：

type: float64  
kind is float64: true  
value: 3.4

还有一些用来修改数据的方法，比如 SetInt、SetFloat。在介绍它们之前，我们要先理解“可修改性”（settability），这一特性会在下面进行详细说明。  
  
反射库提供了很多值得列出来单独讨论的属性，下面就来介绍一下。  
  
首先是介绍下 Value 的 getter 和 setter 方法，为了保证 API 的精简，这两个方法操作的是某一组类型范围最大的那个。比如，处理任何含符号整型数，都使用 int64，也就是说 Value 类型的 Int 方法返回值为 int64 类型，SetInt 方法接收的参数类型也是 int64 类型。实际使用时，可能需要转化为实际的类型：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var x uint8 = 'x'
8. v := reflect.ValueOf(x)
9. fmt.Println("type:", v.Type()) // uint8.
10. fmt.Println("kind is uint8: ", v.Kind() == reflect.Uint8) // true.
11. x = uint8(v.Uint()) // v.Uint returns a uint64.
12. }

运行结果如下：

type: uint8  
kind is uint8: true

其次，反射对象的 Kind 方法描述的是基础类型，而不是静态类型。如果一个反射对象包含了用户定义类型的值，如下所示：

type MyInt int  
var x MyInt = 7  
v := reflect.ValueOf(x)

上面的代码中，虽然变量 v 的静态类型是 MyInt，而不是 int，但 Kind 方法仍然会返回 reflect.Int。换句话说 Kind 方法不会像 Type 方法一样区分 MyInt 和 int。

## 反射第二定律：反射可以将“反射类型对象”转换为“接口类型变量”

和物理学中的反射类似，Go语言中的反射也能创造自己反面类型的对象。  
  
根据一个 reflect.Value 类型的变量，我们可以使用 Interface 方法恢复其接口类型的值。事实上，这个方法会把 type 和 value 信息打包并填充到一个接口变量中，然后返回。  
  
其函数声明如下：

// Interface returns v's value as an interface{}.  
func (v Value) Interface() interface{}

然后，我们可以通过断言，恢复底层的具体值：

y := v.Interface().(float64) // y will have type float64.  
fmt.Println(y)

上面这段代码会打印出一个 float64 类型的值，也就是反射类型变量 v 所代表的值。  
  
事实上，我们可以更好地利用这一特性，标准库中的 fmt.Println 和 fmt.Printf 等函数都接收空接口变量作为参数，fmt 包内部会对接口变量进行拆包，因此 fmt 包的打印函数在打印 reflect.Value 类型变量的数据时，只需要把 Interface 方法的结果传给格式化打印程序：

fmt.Println(v.Interface())

为什么不直接使用 fmt.Println(v)？因为 v 的类型是 reflect.Value，我们需要的是它的具体值，由于值的类型是 float64，我们也可以用浮点格式化打印它：

fmt.Printf("value is %7.1e\n", v.Interface())

运行结果如下：

3.4e+00

同样，这次也不需要对 v.Interface() 的结果进行类型断言，空接口值内部包含了具体值的类型信息，Printf 函数会恢复类型信息。  
  
简单来说 Interface 方法和 ValueOf 函数作用恰好相反，唯一一点是，返回值的静态类型是 interface{}。  
  
Go的反射机制可以将“接口类型的变量”转换为“反射类型的对象”，然后再将“反射类型对象”转换过去。

## 反射第三定律：如果要修改“反射类型对象”其值必须是“可写的”

这条定律很微妙，也很容易让人迷惑，但是如果从第一条定律开始看，应该比较容易理解。  
  
下面这段代码虽然不能正常工作，但是非常值得研究：

var x float64 = 3.4  
v := reflect.ValueOf(x)  
v.SetFloat(7.1) // Error: will panic

如果运行这段代码，它会抛出一个奇怪的异常：

panic: reflect: reflect.flag.mustBeAssignable using unaddressable value

这里问题不在于值7.1 不能被寻址，而是因为变量 v 是“不可写的”，“可写性”是反射类型变量的一个属性，但不是所有的反射类型变量都拥有这个属性。  
  
我们可以通过 CanSet 方法检查一个 reflect.Value 类型变量的“可写性”，对于上面的例子，可以这样写：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var x float64 = 3.4
8. v := reflect.ValueOf(x)
9. fmt.Println("settability of v:", v.CanSet())
10. }

运行结果如下：

settability of v: false

对于一个不具有“可写性”的 Value 类型变量，调用 Set 方法会报出错误。  
  
首先我们要弄清楚什么是“可写性”，“可写性”有些类似于寻址能力，但是更严格，它是反射类型变量的一种属性，赋予该变量修改底层存储数据的能力。“可写性”最终是由一个反射对象是否存储了原始值而决定的。  
  
示例代码如下：

var x float64 = 3.4  
v := reflect.ValueOf(x)

这里我们传递给 reflect.ValueOf 函数的是变量 x 的一个拷贝，而非 x 本身，想象一下如果下面这行代码能够成功执行：

v.SetFloat(7.1)

如果这行代码能够成功执行，它不会更新 x，虽然看起来变量 v 是根据 x 创建的，相反它会更新 x 存在于反射对象 v 内部的一个拷贝，而变量 x 本身完全不受影响。这会造成迷惑，并且没有任何意义，所以是不合法的。“可写性”就是为了避免这个问题而设计的。  
  
这看起来很诡异，事实上并非如此，而且类似的情况很常见。考虑下面这行代码：

f(x)

代码中，我们把变量 x 的一个拷贝传递给函数，因此不期望它会改变 x 的值。如果期望函数 f 能够修改变量 x，我们必须传递 x 的地址（即指向 x 的指针）给函数 f，如下所示：

f(&x)

反射的工作机制与此相同，如果想通过反射修改变量 x，就要把想要修改的变量的指针传递给反射库。  
  
首先，像通常一样初始化变量 x，然后创建一个指向它的反射对象，命名为 p：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var x float64 = 3.4
8. p := reflect.ValueOf(&x) // Note: take the address of x.
9. fmt.Println("type of p:", p.Type())
10. fmt.Println("settability of p:", p.CanSet())
11. }

运行结果如下：

type of p: \*float64  
settability of p: false

反射对象 p 是不可写的，但是我们也不像修改 p，事实上我们要修改的是 \*p。为了得到 p 指向的数据，可以调用 Value 类型的 Elem 方法。Elem 方法能够对指针进行“解引用”，然后将结果存储到反射 Value 类型对象 v 中：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var x float64 = 3.4
8. p := reflect.ValueOf(&x) // Note: take the address of x.
9. v := p.Elem()
10. fmt.Println("settability of v:", v.CanSet())
11. }

运行结果如下：

settability of v: true

由于变量 v 代表 x， 因此我们可以使用 v.SetFloat 修改 x 的值：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var x float64 = 3.4
8. p := reflect.ValueOf(&x) // Note: take the address of x.
9. v := p.Elem()
10. v.SetFloat(7.1)
11. fmt.Println(v.Interface())
12. fmt.Println(x)
13. }

运行结果如下：

7.1  
7.1

反射不太容易理解，reflect.Type 和 reflect.Value 会混淆正在执行的程序，但是它做的事情正是编程语言做的事情。只需要记住：只要反射对象要修改它们表示的对象，就必须获取它们表示的对象的地址。

## 结构体

我们一般使用反射修改结构体的字段，只要有结构体的指针，我们就可以修改它的字段。  
  
下面是一个解析结构体变量 t 的例子，用结构体的地址创建反射变量，再修改它。然后我们对它的类型设置了 typeOfT，并用调用简单的方法迭代字段。  
  
需要注意的是，我们从结构体的类型中提取了字段的名字，但每个字段本身是正常的 reflect.Value 对象。

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. type T struct {
8. A int
9. B string
10. }
11. t := T{23, "skidoo"}
12. s := reflect.ValueOf(&t).Elem()
13. typeOfT := s.Type()
14. for i := 0; i < s.NumField(); i++ {
15. f := s.Field(i)
16. fmt.Printf("%d: %s %s = %v\n", i,
17. typeOfT.Field(i).Name, f.Type(), f.Interface())
18. }
19. }

运行结果如下：

0: A int = 23  
1: B string = skidoo

T 字段名之所以大写，是因为结构体中只有可导出的字段是“可设置”的。  
  
因为 s 包含了一个可设置的反射对象，我们可以修改结构体字段：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. type T struct {
8. A int
9. B string
10. }
11. t := T{23, "skidoo"}
12. s := reflect.ValueOf(&t).Elem()
13. s.Field(0).SetInt(77)
14. s.Field(1).SetString("Sunset Strip")
15. fmt.Println("t is now", t)
16. }

运行结果如下：

t is now {77 Sunset Strip}

如果我们修改了程序让 s 由 t（而不是 &t）创建，程序就会在调用 SetInt 和 SetString 的地方失败，因为 t 的字段是不可设置的。

## 总结

反射规则可以总结为如下几条：

* 反射可以将“接口类型变量”转换为“反射类型对象”；
* 反射可以将“反射类型对象”转换为“接口类型变量”；
* 如果要修改“反射类型对象”，其值必须是“可写的”。

# Go语言反射——性能和灵活性的双刃剑

< [Go语言反射规则浅析](http://c.biancheng.net/view/5131.html)[通过反射获取类型信息](http://c.biancheng.net/view/109.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

现在的一些流行设计思想需要建立在反射基础上，如控制反转（Inversion Of Control，IOC）和依赖注入（Dependency Injection，DI）。Go语言中非常有名的 Web 框架 martini（[https://github.com/go-martini/martini](https://github.com/go-martini/martini" \t "http://c.biancheng.net/view/_blank)）就是通过依赖注入技术进行中间件的实现，例如使用 martini 框架搭建的 http 的服务器如下：

1. package main
2. import "github.com/go-martini/martini"
3. func main() {
4. m := martini.Classic()
5. m.Get("/", func() string {
6. return "Hello world!"
7. })
8. m.Run()
9. }

第 7 行，响应路径/的代码使用一个闭包实现。如果希望获得 Go语言中提供的请求和响应接口，可以直接修改为：

1. m.Get("/", func(res http.ResponseWriter, req \*http.Request) string {
2. // 响应处理代码……
3. })

martini 的底层会自动通过识别 Get 获得的闭包参数情况，通过动态反射调用这个函数并传入需要的参数。martini 的设计广受好评，但同时也有人指出，其运行效率较低。其中最主要的因素是大量使用了反射。  
  
虽然一般情况下，I/O 的延迟远远大于反射代码所造成的延迟。但是，更低的响应速度和更低的 CPU 占用依然是 Web 服务器追求的目标。因此，反射在带来灵活性的同时，也带上了性能低下的桎梏。  
  
要用好反射这把双刃剑，就需要详细了解反射的性能。下面的一些基准测试从多方面对比了原生调用和反射调用的区别。

## 1) 结构体成员赋值对比

反射经常被使用在结构体上，因此结构体的成员访问性能就成为了关注的重点。下面例子中使用一个被实例化的结构体，访问它的成员，然后使用 Go语言的基准化测试可以迅速测试出结果。  
  
反射性能测试的完整代码位于./src/chapter12/reflecttest/reflect\_test.go，下面是对各个部分的详细说明。

本套教程所有源码下载地址：[https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ](https://pan.baidu.com/s/1ORFVTOLEYYqDhRzeq0zIiQ" \t "http://c.biancheng.net/view/_blank)    提取密码：hfyf

原生结构体的赋值过程：

1. // 声明一个结构体, 拥有一个字段
2. type data struct {
3. Hp int
4. }
5. func BenchmarkNativeAssign(b \*testing.B) {
6. // 实例化结构体
7. v := data{Hp: 2}
8. // 停止基准测试的计时器
9. b.StopTimer()
10. // 重置基准测试计时器数据
11. b.ResetTimer()
12. // 重新启动基准测试计时器
13. b.StartTimer()
14. // 根据基准测试数据进行循环测试
15. for i := 0; i < b.N; i++ {
16. // 结构体成员赋值测试
17. v.Hp = 3
18. }
19. }

代码说明如下：

* 第 2 行，声明一个普通结构体，拥有一个成员变量。
* 第 6 行，使用基准化测试的入口。
* 第 9 行，实例化 data 结构体，并给 Hp 成员赋值。
* 第 12～17 行，由于测试的重点必须放在赋值上，因此需要极大程度地降低其他代码的干扰，于是在赋值完成后，将基准测试的计时器复位并重新开始。
* 第 20 行，将基准测试提供的测试数量用于循环中。
* 第 23 行，测试的核心代码：结构体赋值。

接下来的代码分析使用反射访问结构体成员并赋值的过程。

1. func BenchmarkReflectAssign(b \*testing.B) {
2. v := data{Hp: 2}
3. // 取出结构体指针的反射值对象并取其元素
4. vv := reflect.ValueOf(&v).Elem()
5. // 根据名字取结构体成员
6. f := vv.FieldByName("Hp")
7. b.StopTimer()
8. b.ResetTimer()
9. b.StartTimer()
10. for i := 0; i < b.N; i++ {
11. // 反射测试设置成员值性能
12. f.SetInt(3)
13. }
14. }

代码说明如下：

* 第 6 行，取v的地址并转为反射值对象。此时值对象里的类型为 \*data，使用值的 Elem() 方法取元素，获得 data 的反射值对象。
* 第 9 行，使用 FieldByName() 根据名字取出成员的反射值对象。
* 第 11～13 行，重置基准测试计时器。
* 第 18 行，使用反射值对象的 SetInt() 方法，给 data 结构的Hp字段设置数值 3。

这段代码中使用了反射值对象的 SetInt() 方法，这个方法的源码如下：

1. func (v Value) SetInt(x int64) {
2. v.mustBeAssignable()
3. switch k := v.kind(); k {
4. default:
5. panic(&ValueError{"reflect.Value.SetInt", v.kind()})
6. case Int:
7. \*(\*int)(v.ptr) = int(x)
8. case Int8:
9. \*(\*int8)(v.ptr) = int8(x)
10. case Int16:
11. \*(\*int16)(v.ptr) = int16(x)
12. case Int32:
13. \*(\*int32)(v.ptr) = int32(x)
14. case Int64:
15. \*(\*int64)(v.ptr) = x
16. }
17. }

可以发现，整个设置过程都是指针转换及赋值，没有遍历及内存操作等相对耗时的算法。

## 2) 结构体成员搜索并赋值对比

1. func BenchmarkReflectFindFieldAndAssign(b \*testing.B) {
2. v := data{Hp: 2}
3. vv := reflect.ValueOf(&v).Elem()
4. b.StopTimer()
5. b.ResetTimer()
6. b.StartTimer()
7. for i := 0; i < b.N; i++ {
8. // 测试结构体成员的查找和设置成员的性能
9. vv.FieldByName("Hp").SetInt(3)
10. }
11. }

这段代码将反射值对象的 FieldByName() 方法与 SetInt() 方法放在循环里进行检测，主要对比测试 FieldByName() 方法对性能的影响。FieldByName() 方法源码如下：

1. func (v Value) FieldByName(name string) Value {
2. v.mustBe(Struct)
3. if f, ok := v.typ.FieldByName(name); ok {
4. return v.FieldByIndex(f.Index)
5. }
6. return Value{}
7. }

底层代码说明如下：

* 第 3 行，通过名字查询类型对象，这里有一次遍历过程。
* 第 4 行，找到类型对象后，使用 FieldByIndex() 继续在值中查找，这里又是一次遍历。

经过底层代码分析得出，随着结构体字段数量和相对位置的变化，FieldByName() 方法比较严重的低效率问题。

## 3) 调用函数对比

反射的函数调用，也是使用反射中容易忽视的性能点，下面展示对普通函数的调用过程。

1. // 一个普通函数
2. func foo(v int) {
3. }
4. func BenchmarkNativeCall(b \*testing.B) {
5. for i := 0; i < b.N; i++ {
6. // 原生函数调用
7. foo(0)
8. }
9. }
10. func BenchmarkReflectCall(b \*testing.B) {
11. // 取函数的反射值对象
12. v := reflect.ValueOf(foo)
13. b.StopTimer()
14. b.ResetTimer()
15. b.StartTimer()
16. for i := 0; i < b.N; i++ {
17. // 反射调用函数
18. v.Call([]reflect.Value{reflect.ValueOf(2)})
19. }
20. }

代码说明如下：

* 第 2 行，一个普通的只有一个参数的函数。
* 第 10 行，对原生函数调用的性能测试。
* 第 17 行，根据函数名取出反射值对象。
* 第 25 行，使用 reflect.ValueOf(2) 将 2 构造为反射值对象，因为反射函数调用的参数必须全是反射值对象，再使用 []reflect.Value 构造多个参数列表传给反射值对象的 Call() 方法进行调用。

反射函数调用的参数构造过程非常复杂，构建很多对象会造成很大的内存回收负担。Call() 方法内部就更为复杂，需要将参数列表的每个值从 reflect.Value 类型转换为内存。调用完毕后，还要将函数返回值重新转换为 reflect.Value 类型返回。因此，反射调用函数的性能堪忧。

## 4) 基准测试结果对比

测试结果如下：

$ go test -v -bench=.

goos: linux

goarch: amd64

BenchmarkNativeAssign-4 2000000000 0.32 ns/op

BenchmarkReflectAssign-4 300000000 4.42 ns/op

BenchmarkReflectFindFieldAndAssign-4 20000000 91.6 ns/op

BenchmarkNativeCall-4 2000000000 0.33 ns/op

BenchmarkReflectCall-4 10000000 163 ns/op

PASS

结果分析如下：

* 第 4 行，原生的结构体成员赋值，每一步操作耗时 0.32 纳秒，这是参考基准。
* 第 5 行，使用反射的结构体成员赋值，操作耗时 4.42 纳秒，比原生赋值多消耗 13 倍的性能。
* 第 6 行，反射查找结构体成员且反射赋值，操作耗时 91.6 纳秒，扣除反射结构体成员赋值的 4.42 纳秒还富余，性能大概是原生的 272 倍。这个测试结果与代码分析结果很接近。SetInt 的性能可以接受，但 FieldByName() 的性能就非常低。
* 第 7 行，原生函数调用，性能与原生访问结构体成员接近。
* 第 8 行，反射函数调用，性能差到“爆棚”，花费了 163 纳秒，操作耗时比原生多消耗 494 倍。

经过基准测试结果的数值分析及对比，最终得出以下结论：

* 能使用原生代码时，尽量避免反射操作。
* 提前缓冲反射值对象，对性能有很大的帮助。
* 避免反射函数调用，实在需要调用时，先提前缓冲函数参数列表，并且尽量少地使用返回值。

# Go语言reflect.TypeOf()和reflect.Type（通过反射获取类型信息）

< [反射——性能和灵活性的双刃剑](http://c.biancheng.net/view/vip_7358.html)[通过反射获取指针指向的元素类型](http://c.biancheng.net/view/110.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

在 Go语言中通过调用 reflect.TypeOf 函数，我们可以从一个任何非接口类型的值创建一个 reflect.Type 值。reflect.Type 值表示着此非接口值的类型。通过此值，我们可以得到很多此非接口类型的信息。当然，我们也可以将一个接口值传递给一个 reflect.TypeOf 函数调用，但是此调用将返回一个表示着此接口值的动态类型的 reflect.Type 值。  
  
实际上，reflect.TypeOf 函数的唯一参数的类型为 interface{}，reflect.TypeOf 函数将总是返回一个表示着此唯一接口参数值的动态类型的 reflect.Type 值。  
  
那如何得到一个表示着某个接口类型的 reflect.Type 值呢？我们必须通过下面将要介绍的一些间接途径来达到这一目的。  
  
类型 reflect.Type 为一个接口类型，它指定了若干方法（[https://golang.google.cn/pkg/reflect/#Type](https://golang.google.cn/pkg/reflect/" \l "Type" \t "http://c.biancheng.net/view/_blank)）。 通过这些方法，我们能够观察到一个 reflect.Type 值所表示的 Go类型的各种信息。这些方法中的有的适用于所有种类（[https://golang.google.cn/pkg/reflect/#Kind](https://golang.google.cn/pkg/reflect/" \l "Kind" \t "http://c.biancheng.net/view/_blank)）的类型，有的只适用于一种或几种类型。通过不合适的 reflect.Type 属主值调用某个方法将在运行时产生一个恐慌。  
  
使用 reflect.TypeOf() 函数可以获得任意值的类型对象（reflect.Type），程序通过类型对象可以访问任意值的类型信息。下面通过例子来理解获取类型对象的过程：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var a int
8. typeOfA := reflect.TypeOf(a)
9. fmt.Println(typeOfA.Name(), typeOfA.Kind())
10. }

代码输出如下：

int  int

代码说明如下：

* 第 10 行，定义一个 int 类型的变量。
* 第 12 行，通过 reflect.TypeOf() 取得变量 a 的类型对象 typeOfA，类型为 reflect.Type()。
* 第 14 行中，通过 typeOfA 类型对象的成员函数，可以分别获取到 typeOfA 变量的类型名为 int，种类（Kind）为 int。

## 理解反射的类型（Type）与种类（Kind）

在使用反射时，需要首先理解类型（Type）和种类（Kind）的区别。编程中，使用最多的是类型，但在反射中，当需要区分一个大品种的类型时，就会用到种类（Kind）。例如，需要统一判断类型中的指针时，使用种类（Kind）信息就较为方便。

#### 1) 反射种类（Kind）的定义

Go 程序中的类型（Type）指的是系统原生数据类型，如 int、string、bool、float32 等类型，以及使用 type 关键字定义的类型，这些类型的名称就是其类型本身的名称。例如使用 type A struct{} 定义结构体时，A 就是 struct{} 的类型。  
  
种类（Kind）指的是对象归属的品种，在 reflect 包中有如下定义：

1. type Kind uint
2. const (
3. Invalid Kind = iota // 非法类型
4. Bool // 布尔型
5. Int // 有符号整型
6. Int8 // 有符号8位整型
7. Int16 // 有符号16位整型
8. Int32 // 有符号32位整型
9. Int64 // 有符号64位整型
10. Uint // 无符号整型
11. Uint8 // 无符号8位整型
12. Uint16 // 无符号16位整型
13. Uint32 // 无符号32位整型
14. Uint64 // 无符号64位整型
15. Uintptr // 指针
16. Float32 // 单精度浮点数
17. Float64 // 双精度浮点数
18. Complex64 // 64位复数类型
19. Complex128 // 128位复数类型
20. Array // 数组
21. Chan // 通道
22. Func // 函数
23. Interface // 接口
24. Map // 映射
25. Ptr // 指针
26. Slice // 切片
27. String // 字符串
28. Struct // 结构体
29. UnsafePointer // 底层指针
30. )

Map、Slice、Chan 属于引用类型，使用起来类似于指针，但是在种类常量定义中仍然属于独立的种类，不属于 Ptr。  
  
type A struct{} 定义的结构体属于 Struct 种类，\*A 属于 Ptr。

#### 2) 从类型对象中获取类型名称和种类的例子

Go语言中的类型名称对应的反射获取方法是 reflect.Type 中的 Name() 方法，返回表示类型名称的字符串。  
  
类型归属的种类（Kind）使用的是 reflect.Type 中的 Kind() 方法，返回 reflect.Kind 类型的常量。  
  
下面的代码中会对常量和结构体进行类型信息获取。

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. // 定义一个Enum类型
7. type Enum int
8. const (
9. Zero Enum = 0
10. )
11. func main() {
12. // 声明一个空结构体
13. type cat struct {
14. }
15. // 获取结构体实例的反射类型对象
16. typeOfCat := reflect.TypeOf(cat{})
17. // 显示反射类型对象的名称和种类
18. fmt.Println(typeOfCat.Name(), typeOfCat.Kind())
19. // 获取Zero常量的反射类型对象
20. typeOfA := reflect.TypeOf(Zero)
21. // 显示反射类型对象的名称和种类
22. fmt.Println(typeOfA.Name(), typeOfA.Kind())
23. }

代码输出如下：

cat struct  
Enum int

代码说明如下：

* 第 18 行，声明结构体类型 cat。
* 第 22 行，将 cat 实例化，并且使用 reflect.TypeOf() 获取被实例化后的 cat 的反射类型对象。
* 第 25 行，输出cat的类型名称和种类，类型名称就是 cat，而 cat 属于一种结构体种类，因此种类为 struct。
* 第 28 行，Zero 是一个 Enum 类型的常量。这个 Enum 类型在第 9 行声明，第 12 行声明了常量。如没有常量也不能创建实例，通过 reflect.TypeOf() 直接获取反射类型对象。
* 第 31 行，输出 Zero 对应的类型对象的类型名和种类。

# Go语言reflect.Elem()——通过反射获取指针指向的元素类型

< [通过反射获取类型信息](http://c.biancheng.net/view/109.html)[通过反射获取结构体的成员类型](http://c.biancheng.net/view/111.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言程序中对指针获取反射对象时，可以通过 reflect.Elem() 方法获取这个指针指向的元素类型。这个获取过程被称为取元素，等效于对指针类型变量做了一个\*操作，代码如下：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. // 声明一个空结构体
8. type cat struct {
9. }
10. // 创建cat的实例
11. ins := &cat{}
12. // 获取结构体实例的反射类型对象
13. typeOfCat := reflect.TypeOf(ins)
14. // 显示反射类型对象的名称和种类
15. fmt.Printf("name:'%v' kind:'%v'\n",typeOfCat.Name(), typeOfCat.Kind())
16. // 取类型的元素
17. typeOfCat = typeOfCat.Elem()
18. // 显示反射类型对象的名称和种类
19. fmt.Printf("element name: '%v', element kind: '%v'\n", typeOfCat.Name(), typeOfCat.Kind())
20. }

代码输出如下：

name: ''  kind: 'ptr'  
element name: 'cat', element kind: 'struct'

代码说明如下：

* 第 15 行，创建了cat结构体的实例，ins 是一个 \*cat 类型的指针变量。
* 第 18 行，对指针变量获取反射类型信息。
* 第 21 行，输出指针变量的类型名称和种类。Go 语言的反射中对所有指针变量的种类都是 Ptr，但注意，指针变量的类型名称是空，不是 \*cat。
* 第 24 行，取指针类型的元素类型，也就是 cat 类型。这个操作不可逆，不可以通过一个非指针类型获取它的指针类型。
* 第 27 行，输出指针变量指向元素的类型名称和种类，得到了 cat 的类型名称（cat）和种类（struct）。

# Go语言通过反射获取结构体的成员类型

< [通过反射获取指针指向的元素类型](http://c.biancheng.net/view/110.html)[Go语言结构体标签](http://c.biancheng.net/view/112.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

任意值通过 reflect.TypeOf() 获得反射对象信息后，如果它的类型是结构体，可以通过反射值对象（reflect.Type）的 NumField() 和 Field() 方法获得结构体成员的详细信息。与成员获取相关的 reflect.Type 的方法如下表所示。

|  |  |
| --- | --- |
| 结构体成员访问的方法列表 | |
| **方法** | **说明** |
| Field(i int) StructField | 根据索引，返回索引对应的结构体字段的信息。当值不是结构体或索引超界时发生宕机 |
| NumField() int | 返回结构体成员字段数量。当类型不是结构体或索引超界时发生宕机 |
| FieldByName(name string) (StructField, bool) | 根据给定字符串返回字符串对应的结构体字段的信息。没有找到时 bool 返回 false，当类型不是结构体或索引超界时发生宕机 |
| FieldByIndex(index []int) StructField | 多层成员访问时，根据 []int 提供的每个结构体的字段索引，返回字段的信息。没有找到时返回零值。当类型不是结构体或索引超界时 发生宕机 |
| FieldByNameFunc( match func(string) bool) (StructField,bool) | 根据匹配函数匹配需要的字段。当值不是结构体或索引超界时发生宕机 |

## 结构体字段类型

reflect.Type 的 Field() 方法返回 StructField 结构，这个结构描述结构体的成员信息，通过这个信息可以获取成员与结构体的关系，如偏移、索引、是否为匿名字段、结构体标签（Struct Tag）等，而且还可以通过 StructField 的 Type 字段进一步获取结构体成员的类型信息。StructField 的结构如下：

1. type StructField struct {
2. Name string // 字段名
3. PkgPath string // 字段路径
4. Type Type // 字段反射类型对象
5. Tag StructTag // 字段的结构体标签
6. Offset uintptr // 字段在结构体中的相对偏移
7. Index []int // Type.FieldByIndex中的返回的索引值
8. Anonymous bool // 是否为匿名字段
9. }

字段说明如下。

* Name：为字段名称。
* PkgPath：字段在结构体中的路径。
* Type：字段本身的反射类型对象，类型为 reflect.Type，可以进一步获取字段的类型信息。
* Tag：结构体标签，为结构体字段标签的额外信息，可以单独提取。
* Index：FieldByIndex 中的索引顺序。
* Anonymous：表示该字段是否为匿名字段。

## 获取成员反射信息

下面代码中，实例化一个结构体并遍历其结构体成员，再通过 reflect.Type 的 FieldByName() 方法查找结构体中指定名称的字段，直接获取其类型信息。  
  
反射访问结构体成员类型及信息：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. // 声明一个空结构体
8. type cat struct {
9. Name string
10. // 带有结构体tag的字段
11. Type int `json:"type" id:"100"`
12. }
13. // 创建cat的实例
14. ins := cat{Name: "mimi", Type: 1}
15. // 获取结构体实例的反射类型对象
16. typeOfCat := reflect.TypeOf(ins)
17. // 遍历结构体所有成员
18. for i := 0; i < typeOfCat.NumField(); i++ {
19. // 获取每个成员的结构体字段类型
20. fieldType := typeOfCat.Field(i)
21. // 输出成员名和tag
22. fmt.Printf("name: %v tag: '%v'\n", fieldType.Name, fieldType.Tag)
23. }
24. // 通过字段名, 找到字段类型信息
25. if catType, ok := typeOfCat.FieldByName("Type"); ok {
26. // 从tag中取出需要的tag
27. fmt.Println(catType.Tag.Get("json"), catType.Tag.Get("id"))
28. }
29. }

代码输出如下：

name: Name  tag: ''  
name: Type  tag: 'json:"type" id:"100"'  
type 100

代码说明如下：

* 第 11 行，声明了带有两个成员的 cat 结构体。
* 第 15 行，Type 是 cat 的一个成员，这个成员类型后面带有一个以`开始和结尾的字符串。这个字符串在 Go 语言中被称为 Tag（标签）。一般用于给字段添加自定义信息，方便其他模块根据信息进行不同功能的处理。
* 第 19 行，创建 cat 实例，并对两个字段赋值。结构体标签属于类型信息，无须且不能赋值。
* 第 22 行，获取实例的反射类型对象。
* 第 25 行，使用 reflect.Type 类型的 NumField() 方法获得一个结构体类型共有多少个字段。如果类型不是结构体，将会触发宕机错误。
* 第 28 行，reflect.Type 中的 Field() 方法和 NumField 一般都是配对使用，用来实现结构体成员的遍历操作。
* 第 31 行，使用 reflect.Type 的 Field() 方法返回的结构不再是 reflect.Type 而是StructField 结构体。
* 第 35 行，使用 reflect.Type 的 FieldByName() 根据字段名查找结构体字段信息，cat Type 表示返回的结构体字段信息，类型为 StructField，ok 表示是否找到结构体字段的信息。
* 第 38 行中，使用 StructField 中 Tag 的 Get() 方法，根据 Tag 中的名字进行信息获取。

# Go语言结构体标签（Struct Tag）

< [通过反射获取结构体的成员类型](http://c.biancheng.net/view/111.html)[通过反射获取值信息](http://c.biancheng.net/view/113.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

通过 reflect.Type 获取结构体成员信息 reflect.StructField 结构中的 Tag 被称为结构体标签（Struct Tag）。结构体标签是对结构体字段的额外信息标签。  
  
JSON、BSON 等格式进行序列化及对象关系映射（Object Relational Mapping，简称 ORM）系统都会用到结构体标签，这些系统使用标签设定字段在处理时应该具备的特殊属性和可能发生的行为。这些信息都是静态的，无须实例化结构体，可以通过反射获取到。

#### 提示

结构体标签（Struct Tag）类似于 [C#](http://c.biancheng.net/csharp/" \t "http://c.biancheng.net/view/_blank) 中的特性（Attribute）。C# 允许在类、字段、方法等前面添加 Attribute，然后在反射系统中可以获取到这个属性系统。例如：

[Conditional("DEBUG")]  
public static void Message(string msg)  
{  
    Console.WriteLine(msg)；  
}

## 结构体标签的格式

Tag 在结构体字段后方书写的格式如下：

`key1:"value1" key2:"value2"`

结构体标签由一个或多个键值对组成。键与值使用冒号分隔，值用双引号括起来。键值对之间使用一个空格分隔。

## 从结构体标签中获取值

StructTag 拥有一些方法，可以进行 Tag 信息的解析和提取，如下所示：

* func(tag StructTag)Get(key string)string
* 根据 Tag 中的键获取对应的值，例如 `key1:"value1"key2:"value2"` 的 Tag 中，可以传入“key1”获得“value1”。
* func(tag StructTag)Lookup(key string)(value string,ok bool)
* 根据 Tag 中的键，查询值是否存在。

## 结构体标签格式错误导致的问题

编写 Tag 时，必须严格遵守键值对的规则。结构体标签的解析代码的容错能力很差，一旦格式写错，编译和运行时都不会提示任何错误，参见下面这个例子：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. type cat struct {
8. Name string
9. Type int `json: "type" id:"100"`
10. }
11. typeOfCat := reflect.TypeOf(cat{})
12. if catType, ok := typeOfCat.FieldByName("Type"); ok {
13. fmt.Println(catType.Tag.Get("json"))
14. }
15. }

代码输出空字符串，并不会输出期望的 type。  
  
第 12 行中，在json:和"type"之间增加了一个空格。这种写法没有遵守结构体标签的规则，因此无法通过 Tag.Get 获取到正确的 json 对应的值。  
  
这个错误在开发中非常容易被疏忽，造成难以察觉的错误。

# Go语言reflect.ValueOf()和reflect.Value（通过反射获取值信息）

< [Go语言结构体标签](http://c.biancheng.net/view/112.html)[通过反射访问结构体成员的值](http://c.biancheng.net/view/114.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

当我们将一个接口值传递给一个 reflect.ValueOf 函数调用时，此调用返回的是代表着此接口值的动态值的一个 reflect.Value 值。我们必须通过间接的途径获得一个代表一个接口值的 reflect.Value 值。  
  
reflect.Value 类型有很多方法（[https://golang.google.cn/pkg/reflect/](https://golang.google.cn/pkg/reflect/" \t "http://c.biancheng.net/view/_blank)）。我们可以调用这些方法来观察和操纵一个 reflect.Value 属主值表示的 Go 值。这些方法中的有些适用于所有种类类型的值，有些只适用于一种或几种类型的值。  
  
通过不合适的 reflect.Value 属主值调用某个方法将在运行时产生一个恐慌。请阅读 reflect 代码库中各个方法的文档来获取如何正确地使用这些方法。  
  
一个 reflect.Value 值的 CanSet 方法将返回此 reflect.Value 值代表的 Go 值是否可以被修改（可以被赋值）。如果一个 Go 值可以被修改，则我们可以调用对应的 reflect.Value 值的 Set 方法来修改此 Go 值。注意：reflect.ValueOf 函数直接返回的 reflect.Value 值都是不可修改的。  
  
反射不仅可以获取值的类型信息，还可以动态地获取或者设置变量的值。Go语言中使用 reflect.Value 获取和设置变量的值。

## 使用反射值对象包装任意值

Go语言中，使用 reflect.ValueOf() 函数获得值的反射值对象（reflect.Value）。书写格式如下：

value := reflect.ValueOf(rawValue)

reflect.ValueOf 返回 reflect.Value 类型，包含有 rawValue 的值信息。reflect.Value 与原值间可以通过值包装和值获取互相转化。reflect.Value 是一些反射操作的重要类型，如反射调用函数。

## 从反射值对象获取被包装的值

Go语言中可以通过 reflect.Value 重新获得原始值。

#### 1) 从反射值对象（reflect.Value）中获取值的方法

可以通过下面几种方法从反射值对象 reflect.Value 中获取原值，如下表所示。

|  |  |
| --- | --- |
| 反射值获取原始值的方法 | |
| **方法名** | **说  明** |
| Interface() interface {} | 将值以 interface{} 类型返回，可以通过类型断言转换为指定类型 |
| Int() int64 | 将值以 int 类型返回，所有有符号整型均可以此方式返回 |
| Uint() uint64 | 将值以 uint 类型返回，所有无符号整型均可以此方式返回 |
| Float() float64 | 将值以双精度（float64）类型返回，所有浮点数（float32、float64）均可以此方式返回 |
| Bool() bool | 将值以 bool 类型返回 |
| Bytes() []bytes | 将值以字节数组 []bytes 类型返回 |
| String() string | 将值以字符串类型返回 |

#### 2) 从反射值对象（reflect.Value）中获取值的例子

下面代码中，将整型变量中的值使用 reflect.Value 获取反射值对象（reflect.Value）。再通过 reflect.Value 的 Interface() 方法获得 interface{} 类型的原值，通过 int 类型对应的 reflect.Value 的 Int() 方法获得整型值。

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. // 声明整型变量a并赋初值
8. var a int = 1024
9. // 获取变量a的反射值对象
10. valueOfA := reflect.ValueOf(a)
11. // 获取interface{}类型的值, 通过类型断言转换
12. var getA int = valueOfA.Interface().(int)
13. // 获取64位的值, 强制类型转换为int类型
14. var getA2 int = int(valueOfA.Int())
15. fmt.Println(getA, getA2)
16. }

代码输出如下：

1024 1024

代码说明如下：

* 第 11 行，声明一个变量，类型为 int，设置初值为 1024。
* 第 14 行，获取变量 a 的反射值对象，类型为 reflect.Value，这个过程和 reflect.TypeOf() 类似。
* 第 17 行，将 valueOfA 反射值对象以 interface{} 类型取出，通过类型断言转换为 int 类型并赋值给 getA。
* 第 20 行，将 valueOfA 反射值对象通过 Int 方法，以 int64 类型取出，通过强制类型转换，转换为原本的 int 类型。

# Go语言通过反射访问结构体成员的值

< [通过反射获取值信息](http://c.biancheng.net/view/113.html)[判断反射值的空和有效性](http://c.biancheng.net/view/115.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

反射值对象（reflect.Value）提供对结构体访问的方法，通过这些方法可以完成对结构体任意值的访问，如下表所示。

|  |  |
| --- | --- |
| 反射值对象的成员访问方法 | |
| **方  法** | **备  注** |
| Field(i int) Value | 根据索引，返回索引对应的结构体成员字段的反射值对象。当值不是结构体或索引超界时发生宕机 |
| NumField() int | 返回结构体成员字段数量。当值不是结构体或索引超界时发生宕机 |
| FieldByName(name string) Value | 根据给定字符串返回字符串对应的结构体字段。没有找到时返回零值，当值不是结构体或索引超界时发生宕机 |
| FieldByIndex(index []int) Value | 多层成员访问时，根据 []int 提供的每个结构体的字段索引，返回字段的值。 没有找到时返回零值，当值不是结构体或索引超界时发生宕机 |
| FieldByNameFunc(match func(string) bool) Value | 根据匹配函数匹配需要的字段。找到时返回零值，当值不是结构体或索引超界时发生宕机 |

下面代码构造一个结构体包含不同类型的成员。通过 reflect.Value 提供的成员访问函数，可以获得结构体值的各种数据。  
  
反射访问结构体成员值：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. // 定义结构体
7. type dummy struct {
8. a int
9. b string
10. // 嵌入字段
11. float32
12. bool
13. next \*dummy
14. }
15. func main() {
16. // 值包装结构体
17. d := reflect.ValueOf(dummy{
18. next: &dummy{},
19. })
20. // 获取字段数量
21. fmt.Println("NumField", d.NumField())
22. // 获取索引为2的字段(float32字段)
23. floatField := d.Field(2)
24. // 输出字段类型
25. fmt.Println("Field", floatField.Type())
26. // 根据名字查找字段
27. fmt.Println("FieldByName(\"b\").Type", d.FieldByName("b").Type())
28. // 根据索引查找值中, next字段的int字段的值
29. fmt.Println("FieldByIndex([]int{4, 0}).Type()", d.FieldByIndex([]int{4, 0}).Type())
30. }

代码说明如下：

* 第 9 行，定义结构体，结构体的每个字段的类型都不一样。
* 第 24 行，实例化结构体并包装为 reflect.Value 类型，成员中包含一个 \*dummy 的实例。
* 第 29 行，获取结构体的字段数量。
* 第 32 和 35 行，获取索引为2的字段值（float32 字段），并且打印类型。
* 第 38 行，根据b字符串，查找到 b 字段的类型。
* 第 41 行，[]int{4,0} 中的 4 表示，在 dummy 结构中索引值为 4 的成员，也就是 next。next 的类型为 dummy，也是一个结构体，因此使用 []int{4,0} 中的 0 继续在 next 值的基础上索引，结构为 dummy 中索引值为 0 的 a 字段，类型为 int。

代码输出如下：

NumField 5  
Field float32  
FieldByName("b").Type string  
FieldByIndex([]int{4, 0}).Type() int

# Go语言IsNil()和IsValid()——判断反射值的空和有效性

< [通过反射访问结构体成员的值](http://c.biancheng.net/view/114.html)[通过反射修改变量的值](http://c.biancheng.net/view/116.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

反射值对象（reflect.Value）提供一系列方法进行零值和空判定，如下表所示。

|  |  |
| --- | --- |
| 反射值对象的零值和有效性判断方法 | |
| **方 法** | **说 明** |
| IsNil() bool | 返回值是否为 nil。如果值类型不是通道（channel）、函数、接口、map、指针或 切片时发生 panic，类似于语言层的v== nil操作 | |
| IsValid() bool | 判断值是否有效。 当值本身非法时，返回 false，例如 reflect Value不包含任何值，值为 nil 等。 |  |

下面的例子将会对各种方式的空指针进行 IsNil() 和 IsValid() 的返回值判定检测。同时对结构体成员及方法查找 map 键值对的返回值进行 IsValid() 判定，参考下面的代码。  
  
反射值对象的零值和有效性判断：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. // \*int的空指针
8. var a \*int
9. fmt.Println("var a \*int:", reflect.ValueOf(a).IsNil())
10. // nil值
11. fmt.Println("nil:", reflect.ValueOf(nil).IsValid())
12. // \*int类型的空指针
13. fmt.Println("(\*int)(nil):", reflect.ValueOf((\*int)(nil)).Elem().IsValid())
14. // 实例化一个结构体
15. s := struct{}{}
16. // 尝试从结构体中查找一个不存在的字段
17. fmt.Println("不存在的结构体成员:", reflect.ValueOf(s).FieldByName("").IsValid())
18. // 尝试从结构体中查找一个不存在的方法
19. fmt.Println("不存在的结构体方法:", reflect.ValueOf(s).MethodByName("").IsValid())
20. // 实例化一个map
21. m := map[int]int{}
22. // 尝试从map中查找一个不存在的键
23. fmt.Println("不存在的键：", reflect.ValueOf(m).MapIndex(reflect.ValueOf(3)).IsValid())
24. }

代码输出如下：

var a \*int: true  
nil: false  
(\*int)(nil): false  
不存在的结构体成员: false  
不存在的结构体方法: false  
不存在的键： false

代码说明如下：

* 第 11 行，声明一个 \*int 类型的指针，初始值为 nil。
* 第 12 行，将变量 a 包装为 reflect.Value 并且判断是否为空，此时变量 a 为空指针，因此返回 true。
* 第 15 行，对 nil 进行 IsValid() 判定（有效性判定），返回 false。
* 第 18 行，(\*int)(nil) 的含义是将 nil 转换为 \*int，也就是\*int 类型的空指针。此行将 nil 转换为 \*int 类型，并取指针指向元素。由于 nil 不指向任何元素，\*int 类型的 nil 也不能指向任何元素，值不是有效的。因此这个反射值使用 Isvalid() 判断时返回 false。
* 第 21 行，实例化一个结构体。
* 第 24 行，通过 FieldByName 查找 s 结构体中一个空字符串的成员，如成员不存在，IsValid() 返回 false。
* 第 27 行，通过 MethodByName 查找 s 结构体中一个空字符串的方法，如方法不存在，IsValid() 返回 false。
* 第 30 行，实例化一个 map，这种写法与 make 方式创建的 map 等效。
* 第 33 行，MapIndex() 方法能根据给定的 reflect.Value 类型的值查找 map，并且返回查找到的结果。

IsNil() 常被用于判断指针是否为空；IsValid() 常被用于判定返回值是否有效。

# Go语言通过反射修改变量的值

< [判断反射值的空和有效性](http://c.biancheng.net/view/115.html)[通过类型信息创建实例](http://c.biancheng.net/view/117.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言中类似 x、x.f[1] 和 \*p 形式的表达式都可以表示变量，但是其它如 x + 1 和 f(2) 则不是变量。一个变量就是一个可寻址的内存空间，里面存储了一个值，并且存储的值可以通过内存地址来更新。  
  
对于 reflect.Values 也有类似的区别。有一些 reflect.Values 是可取地址的；其它一些则不可以。考虑以下的声明语句：

x := 2 // value type variable?  
a := reflect.ValueOf(2) // 2 int no  
b := reflect.ValueOf(x) // 2 int no  
c := reflect.ValueOf(&x) // &x \*int no  
d := c.Elem() // 2 int yes (x)

其中 a 对应的变量则不可取地址。因为 a 中的值仅仅是整数 2 的拷贝副本。b 中的值也同样不可取地址。c 中的值还是不可取地址，它只是一个指针 &x 的拷贝。实际上，所有通过 reflect.ValueOf(x) 返回的 reflect.Value 都是不可取地址的。但是对于 d，它是 c 的解引用方式生成的，指向另一个变量，因此是可取地址的。我们可以通过调用 reflect.ValueOf(&x).Elem()，来获取任意变量x对应的可取地址的 Value。  
  
我们可以通过调用 reflect.Value 的 CanAddr 方法来判断其是否可以被取地址：

fmt.Println(a.CanAddr()) // "false"  
fmt.Println(b.CanAddr()) // "false"  
fmt.Println(c.CanAddr()) // "false"  
fmt.Println(d.CanAddr()) // "true"

每当我们通过指针间接地获取的 reflect.Value 都是可取地址的，即使开始的是一个不可取地址的 Value。在反射机制中，所有关于是否支持取地址的规则都是类似的。例如，slice 的索引表达式 e[i]将隐式地包含一个指针，它就是可取地址的，即使开始的e表达式不支持也没有关系。  
  
以此类推，reflect.ValueOf(e).Index(i) 对于的值也是可取地址的，即使原始的 reflect.ValueOf(e) 不支持也没有关系。  
  
使用 reflect.Value 对包装的值进行修改时，需要遵循一些规则。如果没有按照规则进行代码设计和编写，轻则无法修改对象值，重则程序在运行时会发生宕机。

## 判定及获取元素的相关方法

使用 reflect.Value 取元素、取地址及修改值的属性方法请参考下表。

|  |  |
| --- | --- |
| 反射值对象的判定及获取元素的方法 | |
| **方法名** | **备  注** |
| Elem() Value | 取值指向的元素值，类似于语言层\*操作。当值类型不是指针或接口时发生宕 机，空指针时返回 nil 的 Value |
| Addr() Value | 对可寻址的值返回其地址，类似于语言层&操作。当值不可寻址时发生宕机 |
| CanAddr() bool | 表示值是否可寻址 |
| CanSet() bool | 返回值能否被修改。要求值可寻址且是导出的字段 |

## 值修改相关方法

使用 reflect.Value 修改值的相关方法如下表所示。

|  |  |
| --- | --- |
| 反射值对象修改值的方法 | |
| **Set(x Value)** | **将值设置为传入的反射值对象的值** |
| Setlnt(x int64) | 使用 int64 设置值。当值的类型不是 int、int8、int16、 int32、int64 时会发生宕机 |
| SetUint(x uint64) | 使用 uint64 设置值。当值的类型不是 uint、uint8、uint16、uint32、uint64 时会发生宕机 |
| SetFloat(x float64) | 使用 float64 设置值。当值的类型不是 float32、float64 时会发生宕机 |
| SetBool(x bool) | 使用 bool 设置值。当值的类型不是 bod 时会发生宕机 |
| SetBytes(x []byte) | 设置字节数组 []bytes值。当值的类型不是 []byte 时会发生宕机 |
| SetString(x string) | 设置字符串值。当值的类型不是 string 时会发生宕机 |

以上方法，在 reflect.Value 的 CanSet 返回 false 仍然修改值时会发生宕机。  
  
在已知值的类型时，应尽量使用值对应类型的反射设置值。

## 值可修改条件之一：可被寻址

通过反射修改变量值的前提条件之一：这个值必须可以被寻址。简单地说就是这个变量必须能被修改。示例代码如下：

1. package main
2. import (
3. "reflect"
4. )
5. func main() {
6. // 声明整型变量a并赋初值
7. var a int = 1024
8. // 获取变量a的反射值对象
9. valueOfA := reflect.ValueOf(a)
10. // 尝试将a修改为1(此处会发生崩溃)
11. valueOfA.SetInt(1)
12. }

程序运行崩溃，打印错误：

panic: reflect: reflect.Value.SetInt using unaddressable value

报错意思是：SetInt 正在使用一个不能被寻址的值。从 reflect.ValueOf 传入的是 a 的值，而不是 a 的地址，这个 reflect.Value 当然是不能被寻址的。将代码修改一下，重新运行：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. // 声明整型变量a并赋初值
8. var a int = 1024
9. // 获取变量a的反射值对象(a的地址)
10. valueOfA := reflect.ValueOf(&a)
11. // 取出a地址的元素(a的值)
12. valueOfA = valueOfA.Elem()
13. // 修改a的值为1
14. valueOfA.SetInt(1)
15. // 打印a的值
16. fmt.Println(valueOfA.Int())
17. }

代码输出如下：

1

下面是对代码的分析：

* 第 14 行中，将变量 a 取值后传给 reflect.ValueOf()。此时 reflect.ValueOf() 返回的 valueOfA 持有变量 a 的地址。
* 第 17 行中，使用 reflect.Value 类型的 Elem() 方法获取 a 地址的元素，也就是 a 的值。reflect.Value 的 Elem() 方法返回的值类型也是 reflect.Value。
* 第 20 行，此时 valueOfA 表示的是 a 的值且可以寻址。使用 SetInt() 方法设置值时不再发生崩溃。
* 第 23 行，正确打印修改的值。

#### 提示

当 reflect.Value 不可寻址时，使用 Addr() 方法也是无法取到值的地址的，同时会发生宕机。虽然说 reflect.Value 的 Addr() 方法类似于语言层的&操作；Elem() 方法类似于语言层的\*操作，但并不代表这些方法与语言层操作等效。

## 值可修改条件之一：被导出

结构体成员中，如果字段没有被导出，即便不使用反射也可以被访问，但不能通过反射修改，代码如下：

1. package main
2. import (
3. "reflect"
4. )
5. func main() {
6. type dog struct {
7. legCount int
8. }
9. // 获取dog实例的反射值对象
10. valueOfDog := reflect.ValueOf(dog{})
11. // 获取legCount字段的值
12. vLegCount := valueOfDog.FieldByName("legCount")
13. // 尝试设置legCount的值(这里会发生崩溃)
14. vLegCount.SetInt(4)
15. }

程序发生崩溃，报错：

panic: reflect: reflect.Value.SetInt using value obtained using unexported field

报错的意思是：SetInt() 使用的值来自于一个未导出的字段。  
  
为了能修改这个值，需要将该字段导出。将 dog 中的 legCount 的成员首字母大写，导出 LegCount 让反射可以访问，修改后的代码如下：

1. type dog struct {
2. LegCount int
3. }

然后根据字段名获取字段的值时，将字符串的字段首字母大写，修改后的代码如下：

1. vLegCount := valueOfDog.FieldByName("LegCount")

再次运行程序，发现仍然报错：

panic: reflect: reflect.Value.SetInt using unaddressable value

这个错误表示第 13 行构造的 valueOfDog 这个结构体实例不能被寻址，因此其字段也不能被修改。修改代码，取结构体的指针，再通过 reflect.Value 的 Elem() 方法取到值的反射值对象。修改后的完整代码如下：

1. package main
2. import (
3. "reflect"
4. "fmt"
5. )
6. func main() {
7. type dog struct {
8. LegCount int
9. }
10. // 获取dog实例地址的反射值对象
11. valueOfDog := reflect.ValueOf(&dog{})
12. // 取出dog实例地址的元素
13. valueOfDog = valueOfDog.Elem()
14. // 获取legCount字段的值
15. vLegCount := valueOfDog.FieldByName("LegCount")
16. // 尝试设置legCount的值(这里会发生崩溃)
17. vLegCount.SetInt(4)
18. fmt.Println(vLegCount.Int())
19. }

代码输出如下：

4

代码说明如下：

* 第 11 行，将 LegCount 首字母大写导出该字段。
* 第 14 行，获取 dog 实例指针的反射值对象。
* 第 17 行，取 dog 实例的指针元素，也就是 dog 的实例。
* 第 20 行，取 dog 结构体中 LegCount 字段的成员值。
* 第 23 行，修改该成员值。
* 第 25 行，打印该成员值。

值的修改从表面意义上叫可寻址，换一种说法就是值必须“可被设置”。那么，想修改变量值，一般的步骤是：

1. 取这个变量的地址或者这个变量所在的结构体已经是指针类型。
2. 使用 reflect.ValueOf 进行值包装。
3. 通过 Value.Elem() 获得指针值指向的元素值对象（Value），因为值对象（Value）内部对象为指针时，使用 set 设置时会报出宕机错误。
4. 使用 Value.Set 设置值。

# Go语言通过类型信息创建实例

< [通过反射修改变量的值](http://c.biancheng.net/view/116.html)[通过反射调用函数](http://c.biancheng.net/view/118.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

当已知 reflect.Type 时，可以动态地创建这个类型的实例，实例的类型为指针。例如 reflect.Type 的类型为 int 时，创建 int 的指针，即\*int，代码如下：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. func main() {
7. var a int
8. // 取变量a的反射类型对象
9. typeOfA := reflect.TypeOf(a)
10. // 根据反射类型对象创建类型实例
11. aIns := reflect.New(typeOfA)
12. // 输出Value的类型和种类
13. fmt.Println(aIns.Type(), aIns.Kind())
14. }

代码输出如下：

\*int ptr

代码说明如下：

* 第 13 行，获取变量 a 的反射类型对象。
* 第 16 行，使用 reflect.New() 函数传入变量 a 的反射类型对象，创建这个类型的实例值，值以 reflect.Value 类型返回。这步操作等效于：new(int)，因此返回的是 \*int 类型的实例。
* 第 19 行，打印 aIns 的类型为 \*int，种类为指针。

# Go语言通过反射调用函数

< [通过类型信息创建实例](http://c.biancheng.net/view/117.html)[Go语言inject库：依赖注入](http://c.biancheng.net/view/5132.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

如果反射值对象（reflect.Value）中值的类型为函数时，可以通过 reflect.Value 调用该函数。使用反射调用函数时，需要将参数使用反射值对象的切片 []reflect.Value 构造后传入 Call() 方法中，调用完成时，函数的返回值通过 []reflect.Value 返回。  
  
下面的代码声明一个加法函数，传入两个整型值，返回两个整型值的和。将函数保存到反射值对象（reflect.Value）中，然后将两个整型值构造为反射值对象的切片（[]reflect.Value），使用 Call() 方法进行调用。  
  
反射调用函数：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. )
6. // 普通函数
7. func add(a, b int) int {
8. return a + b
9. }
10. func main() {
11. // 将函数包装为反射值对象
12. funcValue := reflect.ValueOf(add)
13. // 构造函数参数, 传入两个整型值
14. paramList := []reflect.Value{reflect.ValueOf(10), reflect.ValueOf(20)}
15. // 反射调用函数
16. retList := funcValue.Call(paramList)
17. // 获取第一个返回值, 取整数值
18. fmt.Println(retList[0].Int())
19. }

代码说明如下：

* 第 9～12 行，定义一个普通的加法函数。
* 第 17 行，将 add 函数包装为反射值对象。
* 第 20 行，将 10 和 20 两个整型值使用 reflect.ValueOf 包装为 reflect.Value，再将反射值对象的切片 []reflect.Value 作为函数的参数。
* 第 23 行，使用 funcValue 函数值对象的 Call() 方法，传入参数列表 paramList 调用 add() 函数。
* 第 26 行，调用成功后，通过 retList[0] 取返回值的第一个参数，使用 Int 取返回值的整数值。

#### 提示

反射调用函数的过程需要构造大量的 reflect.Value 和中间变量，对函数参数值进行逐一检查，还需要将调用参数复制到调用函数的参数内存中。调用完毕后，还需要将返回值转换为 reflect.Value，用户还需要从中取出调用值。因此，反射调用函数的性能问题尤为突出，不建议大量使用反射函数调用。

# Go语言inject库：依赖注入

< [通过反射调用函数](http://c.biancheng.net/view/118.html)[Go语言文件处理](http://c.biancheng.net/golang/102/) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

在介绍 inject 之前我们先来简单介绍一下“依赖注入”和“控制反转”这两个概念。  
  
正常情况下，对函数或方法的调用是我们的主动直接行为，在调用某个函数之前我们需要清楚地知道被调函数的名称是什么，参数有哪些类型等等。  
  
所谓的控制反转就是将这种主动行为变成间接的行为，我们不用直接调用函数或对象，而是借助框架代码进行间接的调用和初始化，这种行为称作“控制反转”，库和框架能很好的解释控制反转的概念。  
  
依赖注入是实现控制反转的一种方法，如果说控制反转是一种设计思想，那么依赖注入就是这种思想的一种实现，通过注入参数或实例的方式实现控制反转。如果没有特殊说明，我们可以认为依赖注入和控制反转是一个东西。  
  
控制反转的价值在于解耦，有了控制反转就不需要将代码写死，可以让控制反转的的框架代码读取配置，动态的构建对象，这一点在 [Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) 的 [Spring](http://c.biancheng.net/spring/" \t "http://c.biancheng.net/view/_blank) 框架中体现的尤为突出。

## inject 实践

inject 是依赖注入的Go语言实现，它能在运行时注入参数，调用方法，是 Martini 框架（Go语言中著名的 Web 框架）的基础核心。  
  
在介绍具体实现之前，先来想一个问题，如何通过一个字符串类型的函数名来调用函数？Go语言没有 Java 中的 Class.forName 方法可以通过类名直接构造对象，所以这种方法是行不通的，能想到的方法就是使用 map 实现一个字符串到函数的映射，示例代码如下：

1. func fl() {
2. println ("fl")
3. }
4. func f2 () {
5. println ("f2")
6. }
7. funcs := make(map[string] func ())
8. funcs ["fl"] = fl
9. funcs ["f2"] = fl
10. funcs ["fl"]()
11. funcs ["f2"]()

但是这有个缺陷，就是 map 的 Value 类型被写成 func()，不同参数和返回值的类型的函数并不能通用。将 map 的 Value 定义为 interface{} 空接口类型即可以解决该问题，但需要借助类型断言或反射来实现，通过类型断言实现等于又绕回去了，反射是一种可行的办法。  
  
inject 包借助反射实现函数的注入调用，下面通过一个示例来看一下。

1. package main
2. import (
3. "fmt"
4. "github.com/codegangsta/inject"
5. )
6. type S1 interface{}
7. type S2 interface{}
8. func Format(name string, company S1, level S2, age int) {
9. fmt.Printf("name ＝ %s, company=%s, level=%s, age ＝ %d!\n", name, company, level, age)
10. }
11. func main() {
12. //控制实例的创建
13. inj := inject.New()
14. //实参注入
15. inj.Map("tom")
16. inj.MapTo("tencent", (\*S1)(nil))
17. inj.MapTo("T4", (\*S2)(nil))
18. inj.Map(23)
19. //函数反转调用
20. inj.Invoke(Format)
21. }

运行结果如下：

name ＝ tom, company=tencent, level=T4, age ＝ 23!

可见 inject 提供了一种注入参数调用函数的通用功能，inject.New() 相当于创建了一个控制实例，由其来实现对函数的注入调用。inject 包不但提供了对函数的注入，还实现了对 struct 类型的注入，示例代码如下所示：

1. package main
2. import (
3. "fmt"
4. "github.com/codegangsta/inject"
5. )
6. type S1 interface{}
7. type S2 interface{}
8. type Staff struct {
9. Name string `inject`
10. Company S1 `inject`
11. Level S2 `inject`
12. Age int `inject`
13. }
14. func main() {
15. //创建被注入实例
16. s := Staff{}
17. //控制实例的创建
18. inj := inject.New()
19. //初始化注入值
20. inj.Map("tom")
21. inj.MapTo("tencent", (\*S1)(nil))
22. inj.MapTo("T4", (\*S2)(nil))
23. inj.Map(23)
24. //实现对 struct 注入
25. inj.Apply(&s)
26. //打印结果
27. fmt.Printf("s ＝ %v\n", s)
28. }

运行结果如下：

s ＝ {tom tencent T4 23}

可以看到 inject 提供了一种对结构类型的通用注入方法。至此，我们仅仅从宏观层面了解 iniect 能做什么，下面从源码实现角度来分析 inject。

## inject 原理分析

inject 包中只有 2 个文件，一个是 inject.go 文件和一个 inject\_test.go 文件，这里我们只需要关注 inject.go 文件即可。  
  
inject.go 短小精悍，包括注释和空行在内才 157 行代码，代码中定义了 4 个接口，包括一个父接口和三个子接口，如下所示：

1. type Injector interface {
2. Applicator
3. Invoker
4. TypeMapper
5. SetParent(Injector)
6. }
7. type Applicator interface {
8. Apply(interface{}) error
9. }
10. type Invoker interface {
11. Invoke(interface{}) ([]reflect.Value, error)
12. }
13. type TypeMapper interface {
14. Map(interface{}) TypeMapper
15. MapTo(interface{}, interface{}) TypeMapper
16. Get(reflect.Type) reflect.Value
17. }

Injector 接口是 Applicator、Invoker、TypeMapper 接口的父接口，所以实现了 Injector 接口的类型，也必然实现了 Applicator、Invoker 和 TypeMapper 接口：

* Applicator 接口只规定了 Apply 成员，它用于注入 struct。
* Invoker 接口只规定了 Invoke 成员，它用于执行被调用者。
* TypeMapper 接口规定了三个成员，Map 和 MapTo 都用于注入参数，但它们有不同的用法，Get 用于调用时获取被注入的参数。

另外 Injector 还规定了 SetParent 行为，它用于设置父 Injector，其实它相当于查找继承。也即通过 Get 方法在获取被注入参数时会一直追溯到 parent，这是个递归过程，直到查找到参数或为 nil 终止。

1. type injector struct {
2. values map[reflect.Type]reflect.Value
3. parent Injector
4. }
5. func InterfaceOf(value interface{}) reflect.Type {
6. t := reflect.TypeOf(value)
7. for t.Kind() == reflect.Ptr {
8. t = t.Elem()
9. }
10. if t.Kind() != reflect.Interface {
11. panic("Called inject.InterfaceOf with a value that is not a pointer to an interface. (\*MyInterface)(nil)")
12. }
13. return t
14. }
15. func New() Injector {
16. return &injector{
17. values: make(map[reflect.Type]reflect.Value),
18. }
19. }

injector 是 inject 包中唯一定义的 struct，所有的操作都是基于 injector struct 来进行的，它有两个成员 values 和 parent。values 用于保存注入的参数，是一个用 reflect.Type 当键、reflect.Value 为值的 map，理解这点将有助于理解 Map 和 MapTo。  
  
New 方法用于初始化 injector struct，并返回一个指向 injector struct 的指针，但是这个返回值被 Injector 接口包装了。  
  
InterfaceOf 方法虽然只有几句实现代码，但它是 Injector 的核心。InterfaceOf 方法的参数必须是一个接口类型的指针，如果不是则引发 panic。InterfaceOf 方法的返回类型是 reflect.Type，大家应该还记得 injector 的成员 values 就是一个 reflect.Type 类型当键的 map。这个方法的作用其实只是获取参数的类型，而不关心它的值。  
  
示例代码如下所示：

1. package main
2. import (
3. "fmt"
4. "github.com/codegangsta/inject"
5. )
6. type SpecialString interface{}
7. func main() {
8. fmt.Println(inject.InterfaceOf((\*interface{})(nil)))
9. fmt.Println(inject.InterfaceOf((\*SpecialString)(nil)))
10. }

运行结果如下：

interface {}  
main.SpecialString

InterfaceOf 方法就是用来得到参数类型，而不关心它具体存储的是什么值。

1. func (i \*injector) Map(val interface{}) TypeMapper {
2. i.values[reflect.TypeOf(val)] = reflect.ValueOf(val)
3. return i
4. }
5. func (i \*injector) MapTo(val interface{}, ifacePtr interface{}) TypeMapper {
6. i.values[InterfaceOf(ifacePtr)] = reflect.ValueOf(val)
7. return i
8. }
9. func (i \*injector) Get(t reflect.Type) reflect.Value {
10. val := i.values[t]
11. if !val.IsValid() && i.parent != nil {
12. val = i.parent.Get(t)
13. }
14. return val
15. }
16. func (i \*injector) SetParent(parent Injector) {
17. i.parent = parent
18. }

Map 和 MapTo 方法都用于注入参数，保存于 injector 的成员 values 中。这两个方法的功能完全相同，唯一的区别就是 Map 方法用参数值本身的类型当键，而 MapTo 方法有一个额外的参数可以指定特定的类型当键。但是 MapTo 方法的第二个参数 ifacePtr 必须是接口指针类型，因为最终 ifacePtr 会作为 InterfaceOf 方法的参数。  
  
为什么需要有 MapTo 方法？因为注入的参数是存储在一个以类型为键的 map 中，可想而知，当一个函数中有一个以上的参数的类型是一样时，后执行 Map 进行注入的参数将会覆盖前一个通过 Map 注入的参数。  
  
SetParent 方法用于给某个 Injector 指定父 Injector。Get 方法通过 reflect.Type 从 injector 的 values 成员中取出对应的值，它可能会检查是否设置了 parent，直到找到或返回无效的值，最后 Get 方法的返回值会经过 IsValid 方法的校验。  
  
示例代码如下所示：

1. package main
2. import (
3. "fmt"
4. "reflect"
5. "github.com/codegangsta/inject"
6. )
7. type SpecialString interface{}
8. func main() {
9. inj := inject.New()
10. inj.Map("C语言中文网")
11. inj.MapTo("Golang", (\*SpecialString)(nil))
12. inj.Map(20)
13. fmt.Println("字符串是否有效？", inj.Get(reflect.TypeOf("Go语言入门教程")).IsValid())
14. fmt.Println("特殊字符串是否有效？", inj.Get(inject.InterfaceOf((\*SpecialString)(nil))).IsValid())
15. fmt.Println("int 是否有效？", inj.Get(reflect.TypeOf(18)).IsValid())
16. fmt.Println("[]byte 是否有效？", inj.Get(reflect.TypeOf([]byte("Golang"))).IsValid())
17. inj2 := inject.New()
18. inj2.Map([]byte("test"))
19. inj.SetParent(inj2)
20. fmt.Println("[]byte 是否有效？", inj.Get(reflect.TypeOf([]byte("Golang"))).IsValid())
21. }

运行结果如下所示：

字符串是否有效？ true  
特殊字符串是否有效？ true  
int 是否有效？ true  
[]byte 是否有效？ false  
[]byte 是否有效？ true

通过以上例子应该知道 SetParent 是什么样的行为，是不是很像面向对象中的查找链？

1. func (inj \*injector) Invoke(f interface{}) ([]reflect.Value, error) {
2. t := reflect.TypeOf(f)
3. var in = make([]reflect.Value, t.NumIn()) //Panic if t is not kind of Func
4. for i := 0; i < t.NumIn(); i++ {
5. argType := t.In(i)
6. val := inj.Get(argType)
7. if !val.IsValid() {
8. return nil, fmt.Errorf("Value not found for type %v", argType)
9. }
10. in[i] = val
11. }
12. return reflect.ValueOf(f).Call(in), nil
13. }

Invoke 方法用于动态执行函数，当然执行前可以通过 Map 或 MapTo 来注入参数，因为通过 Invoke 执行的函数会取出已注入的参数，然后通过 reflect 包中的 Call 方法来调用。Invoke 接收的参数 f 是一个接口类型，但是 f 的底层类型必须为 func，否则会 panic。

1. package main
2. import (
3. "fmt"
4. "github.com/codegangsta/inject"
5. )
6. type SpecialString interface{}
7. func Say(name string, gender SpecialString, age int) {
8. fmt.Printf("My name is %s, gender is %s, age is %d!\n", name, gender, age)
9. }
10. func main() {
11. inj := inject.New()
12. inj.Map("张三")
13. inj.MapTo("男", (\*SpecialString)(nil))
14. inj2 := inject.New()
15. inj2.Map(25)
16. inj.SetParent(inj2)
17. inj.Invoke(Say)
18. }

运行结果如下：

My name is 张三, gender is 男, age is 25!

上面的例子如果没有定义 SpecialString 接口作为 gender 参数的类型，而把 name 和 gender 都定义为 string 类型，那么 gender 会覆盖 name 的值。

1. func (inj \*injector) Apply(val interface{}) error {
2. v := reflect.ValueOf(val)
3. for v.Kind() == reflect.Ptr {
4. v = v.Elem()
5. }
6. if v.Kind() != reflect.Struct {
7. return nil
8. }
9. t := v.Type()
10. for i := 0; i < v.NumField(); i++ {
11. f := v.Field(i)
12. structField := t.Field(i)
13. if f.CanSet() && structField.Tag == "inject" {
14. ft := f.Type()
15. v := inj.Get(ft)
16. if !v.IsValid() {
17. return fmt.Errorf("Value not found for type %v", ft)
18. }
19. f.Set(v)
20. }
21. }
22. return nil
23. }

Apply 方法是用于对 struct 的字段进行注入，参数为指向底层类型为结构体的指针。可注入的前提是：字段必须是导出的（也即字段名以大写字母开头），并且此字段的 tag 设置为`inject`。  
  
示例代码如下所示：

1. package main
2. import (
3. "fmt"
4. "github.com/codegangsta/inject"
5. )
6. type SpecialString interface{}
7. type TestStruct struct {
8. Name string `inject`
9. Nick []byte
10. Gender SpecialString `inject`
11. uid int `inject`
12. Age int `inject`
13. }
14. func main() {
15. s := TestStruct{}
16. inj := inject.New()
17. inj.Map("张三")
18. inj.MapTo("男", (\*SpecialString)(nil))
19. inj2 := inject.New()
20. inj2.Map(26)
21. inj.SetParent(inj2)
22. inj.Apply(&s)
23. fmt.Println("s.Name =", s.Name)
24. fmt.Println("s.Gender =", s.Gender)
25. fmt.Println("s.Age =", s.Age)
26. }

运行结果如下：

s.Name = 张三  
s.Gender = 男  
s.Age = 26